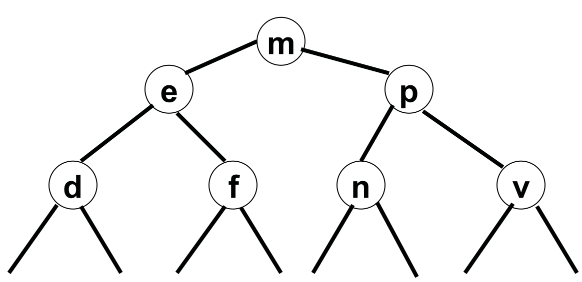
1.00 Lecture 37

Data Structures:
TreeMap, HashMap

Binary Trees



Level	Nodes
0	2^0
1	2^1
2	2^2
…	
k	2^k

- Full binary tree has $2^{(k+1)}$-1 nodes
- Maximum of k steps required to find (or not find) a node
  - E.g. 2^{20} nodes, or 1,000,000 nodes, in 20 steps!
- In a binary search tree (but not all types of binary tree):
  - All nodes to left are smaller than parent
  - All nodes to right are larger than parent
  - No ties: each node has a unique key or id

Exercise: Binary Search Tree, Adding Nodes

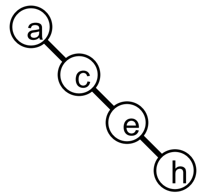
- Start with an empty binary search tree.
- Insert the following nodes while maintaining the binary search tree property:
  - `"b", "q", "t", "d", "a"`
- The first node, "b", will be the root.
- Where will the second node, "q", go?
- Draw the tree that results with all 5 nodes

Binary Search Tree (BST)

- Binary search trees are used to store large amounts of data
  - High capacity (~2^k)
  - Fast access (k steps)
- Basic tree operations (insert, find, delete) are not difficult to implement
  - Special cases take some care, as in all data structures
  - And keeping the tree balanced, so that all branches are of comparable length, requires sophistication
  - We won't implement any tree code; we'll use the Java implementation

Java Tree Implementation

- Trees are efficient if they are <u>balanced</u>
  - A balanced tree of depth 20 can hold about 2^{20}, or 1,000,000 nodes
  - If the tree were unbalanced, in the worst case it would require 1,000,000 levels to hold 1,000,000 nodes, and 1,000,000 steps to find/insert/delete



- To prevent unbalance, Java uses a sophisticated binary tree called known as a red-black tree.
  - Red-black trees automatically rebalance themselves if one branch becomes deeper than a sibling.
  - Other, similar algorithms include AVL trees, 2-3 trees, …

Keys, Sets, and Maps

- Every node in a tree has a *key*, which is a unique identifier
  - If a node contains nothing but the *key*, it is called a `TreeSet`
  - Transit example: gate at Kendall Sq has tree of CharlieCard numbers
  - If a node contains a *key* and a *value,* it is called a `TreeMap`.
  - Phone book example: *key*= your name, *value*= your phone number
  - Trees keep nodes in a defined order, as in a phone book (alphabetical)
- The *key* is used to look up the *value*.
  - The *value* is extra data contained in the node indexed by the *key*.
  - Nodes <u>must</u> have unique keys to distinguish between them.
- Typical methods in a `TreeSet<Integer>` are:
  - `boolean contains(Integer n)`
  - `Integer first()`
- The equivalent methods in a `TreeMap<Integer, String>` are:
  - `boolean containsKey(Integer n)`
  - `String get(Integer n)`
  - `Integer firstKey()`
  - `String firstValue()`

How to Traverse a `TreeMap`

Given a `TreeMap<Integer, String>`, how would you print out every entry in order?

```
TreeMap<Integer, String> list=
  new TreeMap<Integer, String> ();
// add entries
for (Integer n : list.keySet()) {
  System.out.println( n + ", " +
    list.get(n));
}
```

`Comparable<T>`

- **Recall the Comparable interface from sorting**
- **In trees, all keys must belong to a single class that implements the `Comparable<T>` interface**
  - **Or you can supply a `Comparator<T>` to the constructor**
- **`Comparable<T>` has one method**:

  `int compareTo(T other)`
  - **`compareTo` returns:**
    - **An int < 0 if (this < other)**
    - **0 if (other equals this)**
    - **An int > 0 if (this > other)**
- **Many Java classes already implement `Comparable`, e.g. `String, Integer`**

Exercise 1: TreeSet

```
public class FullName implements Comparable<FullName> {
    private final String firstName;
    private final String lastName;

    public FullName( String f, String l ) {
      firstName= f;
      lastName= l;
    }

    public String getFirstName() {return firstName;}
    public String getLastName()  {return lastName;}

    public int compareTo( FullName fn ) {
      // Complete the  compareTo() method
      // Order by last name, then first name
      // Remember String has a compareTo() method already
      // You are comparing pairs of Strings
    }

    public String toString() {
      return firstName + " " + lastName;
    }
}
```

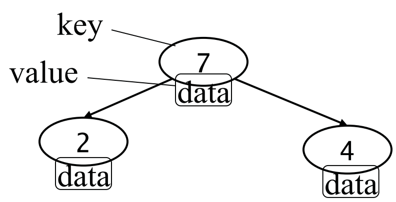
Exercise 1, p.2

```
import java.util.*;

public class FullNameTest {

  public static void main(String[] args) {
    FullName scott= new FullName("Scott", "Stevens");
    FullName ellen= new FullName("Ellen", "Shipps");
    FullName andrea= new FullName("Andrea", "Kondoleon");
    FullName paul= new FullName("Paul", "Stevens");

    TreeSet<FullName> names= new TreeSet<FullName>();
    names.add(scott);
    names.add(ellen);
    names.add(andrea);
    names.add(paul);

    for (FullName1 f : names)
       System.out.println(f);
  }
}
```

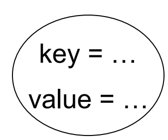
Keys and Values

What good is a tree of numbers?

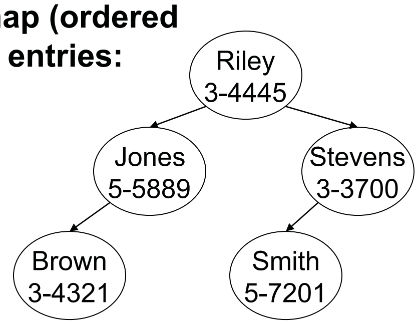
- **A "key" in a tree is an ordered value, i.e. a key can be compared with another object of the same type**
- **A node in an ordered binary tree consists of an ordered key and a value**
- **All the keys in a tree should be of the same type**



Tree Map



- **Each node has a key and a value**
- **Phonebook example:**
  - **Key: name**
  - **Value: phone number**
- **Exercise: Draw the tree map (ordered alphabetically) with these entries:**
  - **Riley, 3-4445**
  - **Stevens, 3-3700**
  - **Smith, 5-7201**
  - **Jones, 5-5889**
  - **Brown, 3-4321**



Exercise 2: TreeMap

- **We use a `TreeMap<FullName, String>` to create a phone book; this code is provided in class PhoneBook**
  - **FullName is the key; String (phone number) is the value**
- **We use a loop to display a `JOptionPane` that asks for a full name, in the format: "firstName lastName"**
  - **Your code will try to look up the phone number for this name**
- **Use the `String` method `split()` to parse the name**
  - **split() takes the delimiter as its argument, e.g., " " here (space)**
  - **split() returns an array of Strings**
- **Use the `TreeMap<FullName,String>` method**

  `String get(FullName fn)`

  to return the subscriber entry.
  - **get() will return the value if the key is found**
  - **get() will return `null` if the key cannot be found.**

PhoneBook.java

```
import java.util.*;
import javax.swing.JOptionPane;

public class PhoneBook {
  public static void main(String[] args) {
    FullName1 scott= new FullName1("Scott", "Stevens");
    FullName1 ellen= new FullName1("Ellen", "Shipps");
    FullName1 pizza= new FullName1("Michael", "Pizza");
    FullName1 paul= new FullName1("Paul", "Stevens");
    TreeMap<FullName1,String> phones=
      new TreeMap<FullName1,String>();

    phones.put(scott, "617-225-7178");
    phones.put(ellen, "781-646-2880");
    phones.put(pizza, "781-648-2000");
    phones.put(paul, "617-498-2142");
```

PhoneBook.java, p.2

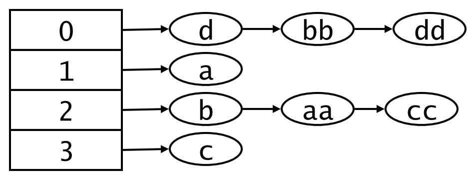
```
    while (true) {
      String text= JOptionPane.showInputDialog(
        "Enter full name");
      if (text.isEmpty())
             break;
      // Your code here
      // Parse the full name (“firstName lastName”)
      // Use the get() method with FullName1 key to retrieve
      //   the String phone number value.
      // Print out the phone number or “Subscriber unknown”
      //   if get() returns null
    }
  }
}
```

Exercise 3: Data Structure Efficiency

- If you are searching an unordered list of n items for an element, on average how many items will you have to search to find the item:
  - If item is present in the list?
  - If item is not present in the list?
- What happens if the list is ordered?
  - If item is present in the list?
  - If item is not present in the list?
- If the items are stored in a TreeMap how many items will you have to search on average?
  - Whether item is present or not
- Can we do better?

Hashing Illustration

```
keys = { a, b, c, d, aa, bb, cc, dd }
Hash function: (sum of chars) % 4
a= 97, b= 98, c= 99, d= 100
```



Hash table (hash map)

- **Hashing maps each Object to an index in an array of Node references**
- **The array contains the "first" reference to a linked list of Objects.**
- **We traverse the list to add or find Objects that hash to that value**
- **We keep the lists short, so hash efficiency is close to array index lookup**

`HashMap`

- **`HashMap` holds keys and values, similar to `TreeMap`**
  - **HashMap like a filing cabinet, in which each folder has a tab (hash code) and contains a small number of objects (list)**
- **HashMap provides constant time lookup no matter how many elements it contains.**
  - **If we have n= 1,000,000 items, and t is the time to find one item, then**
    - **`LinkedList` will take ~500,000t (n/2) to find an item;**
    - **`TreeMap` will take ~20t ($\log_2$ n)**
    - **`HashMap` will take ~t**
- **Lookup time depends on having a good `hashCode()` method and is statistical.**
- **Elements in a `HashMap` are NOT ordered by anything useful. Storage order is by hash code.**

Java hashCode()

- **Hashing is done in two phases**
  - ***hash1* function is responsibility of the key class (the data type being stored in the hash table)**
  - **In the example, *hash1* is (sum of characters)**
  - ***hash2* function is responsibility of hash table: it takes *hash1* and maps it to the number of slots in the hash table**
  - **In the example, *hash2* is (% 4)**
- **Hash table class does not know enough to generate a hash code from a particular object**
  - ***hash1* should map objects as evenly as possible to different hash values**
- **Java `Object` has `int hashCode()` method**
  - **Thus all Java objects inherit hashCode()**
  - **Caution: default `hashCode()` method can return a negative integer**
  - **We usually take the absolute value of the `hashCode()`**

hashCode()

- **All `hashCode()` methods must return the same integer when presented with the same object**

```
if ( o1.equals( o2 ))
  o1.hashCode()==o2.hashCode // must be true
```

  - **They cannot return a random integer.**
  - **If they did, there would be no way to look up a key once it had been inserted in the hash table**
- **All `hashCode()` methods should return a different integer from a different object**
- **Java classes (`String, JButton`, etc.) have good `hashCode()` methods.**
- **`Object` has a terrible `hashCode()` method.**
  - **If you extend `Object` and you intend to use the new class as a key in a `HashMap`, you should override the `hashCode()` method**

equals()

- `equals()` method returns `true` if two objects are equivalent
  - `Object` class default `equals()` tests if two objects are at same memory location. It doesn't look at their fields.
- Better `equals()` than default needed in `HashMaps` to find matching objects
- `equals()` method in `MyClass` should use the following pattern:

```
public boolean equals(Object other) {
  if (this == other) return true;
  if (!(other instanceof MyClass) ) return false;
  MyClass otherOne= (MyClass) other;
  if ( /*this and otherOne have equivalent fields*/)
    return true;
  else return false;
}
```

Exercise 4: HashPhoneBook

- `FullName` is the same as used in `TreeMap`.
- Copy and rename your `PhoneBook` solution to `HashPhoneBook`
  - `HashPhoneBook` is the same as `PhoneBook` except it uses a `HashMap` instead of a `TreeMap`.
  - Make that one change in your code
- Run `HashPhoneBook`
  - Can you find any subscribers' numbers? Why not?
- Fix the problem by adding good `equals()` and `hashCode()` methods to `FullName`.
  - Use the pattern from the previous slide for equals()
  - Use the String hashCode() method within your hashCode()
- Test using `HashPhoneBook`.

Exercise 4

```
public class FullName
  implements Comparable<FullName> {
    private final String firstName;
    private final String lastName;

  public FullName( String f, String l ) {
    firstName= f; lastName= l;
  }

  …
  // Add overrides for versions in Object for:
  //   public boolean equals( Object o )
  //   public int hashCode()
}
```

Exercise 5: Which Data Structure?

- In programs that must store and retrieve large amounts of data, you typically choose between TreeMap and HashMap
  - Hashing is faster but does not keep Objects in order
  - Trees are slower but keep Objects in order
- In each of the following cases, select the data structure(s) most appropriate to the application.
  - Planes, runways and gates in a simulation
  - Books in a library
  - Airline reservations for many passengers
  - Events in a data communication system

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012